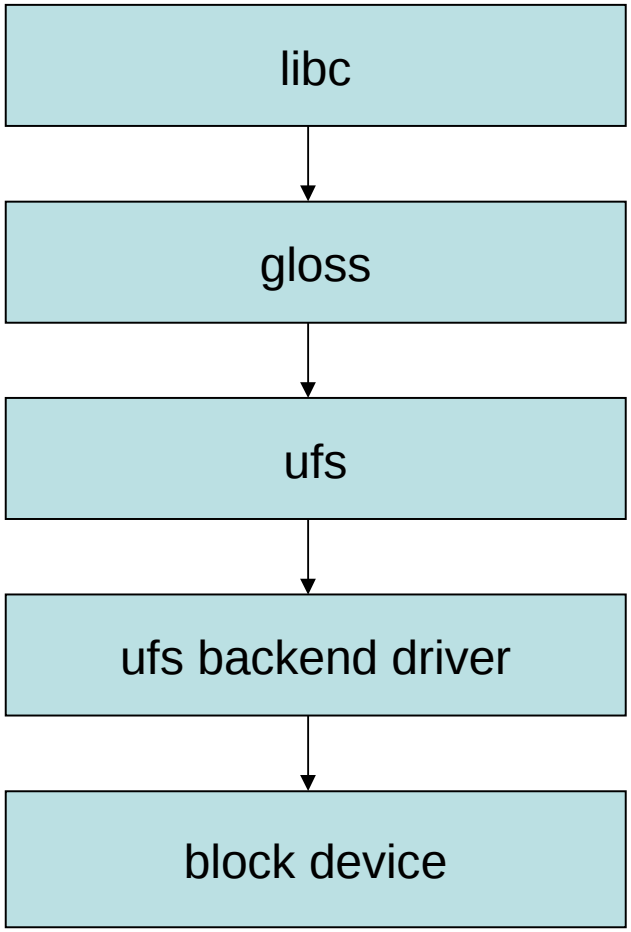
Inside File IO

Introduction to Marvell SDK File IO Architecture

Walter Zhou, walterzh@marvell.com

2015-01-03


libc
gloss
ufs
ufs backend driver
block device

File IO in libc

in Makefile

MODIFIED_C := $(BUILD_ROOT)/**libmodified_c.a**

STANDARD_C := $(STANDARD_LIBRARY_LOCATION)/**libc.a**

$(MODIFIED_C): $(STANDARD_C)

    @echo Deleting symbols from libc.a

    $(CP) $(STANDARD_C) $(MODIFIED_C)

    $(AR) d $(MODIFIED_C) lib_a-mallocr.o lib_a-reallocr.o lib_a-callocr.o lib_a-freer.o lib_a-syscalls.o lib_a-signalr.o lib_a-signal.o

$(APPTARGET).elf: fw_version_$(PRODUCT).o $(BUILTIN_ROMFILES_OBJ) **$(MODIFIED_C)**

$(call apptarget_rules)

File IO in libc

walterzh@ T1650: ~/temp.libc$ ar t libmodified_c.a
lib_a-fopen.o
lib_a-fprintf.o
lib_a-fputc.o
lib_a-fputs.o
lib_a-fread.o
lib_a-freopen.o
lib_a-fscanf.o
lib_a-fseek.o
lib_a-fseeko.o
lib_a-fsetpos.o
lib_a-fstatr.o
lib_a-ftell.o
lib_a-ftello.o
lib_a-fvwrite.o
lib_a-fwalk.o
lib_a-fwrite.o
lib_a-gdtoa-gethex.o
……

gloss

The module is for binding libc and ufs.

1. The backend of libc (like system call in Unix)
2. Make libc happy

gloss

The backend of libc (like system call in Unix)

For example:

in gloss_write.c file

ssize_t _write(int fd, const void *buf, size_t cnt)

The function is invoked by libc, not for other modules!

When you invoke write() or fwrite() APIs in libc, they will be routed to the function.

gloss

Make libc happy

For example:

in Gloss.c file

```
int getpid(void)
{
    REL_ASSERT(0);
    return 1;
}

uid_t getuid(void) { return 0; }
uid_t geteuid(void) { return 0; }
uid_t getgid(void) { return 0; }
uid_t getegid(void) { return 0; }
```

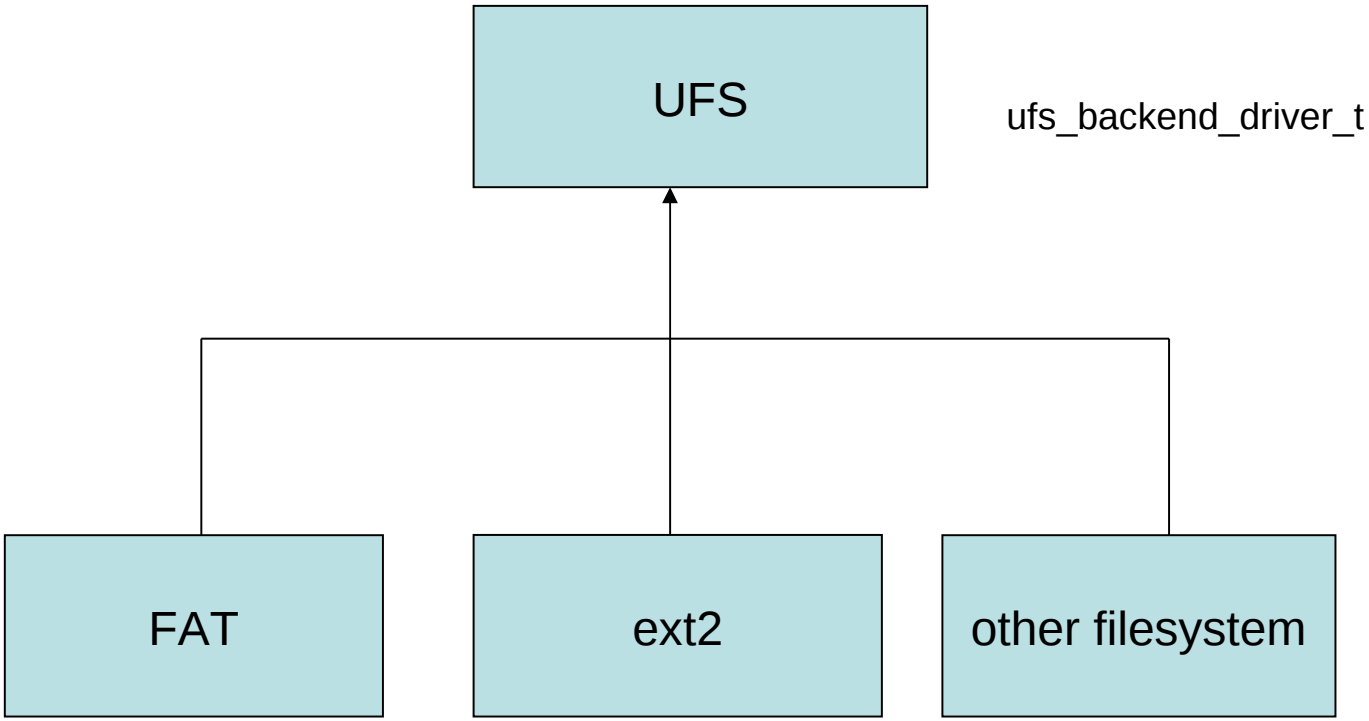
UFS

ufs --- **u**nified **f**ile **s**ystem

It's like vfs (Virtual File System or Virtual File Switch in Linux).

You could think ufs is a pure virtual base class, and the real file system is its sub-class.

UFS



UFS

```
typedef struct
{
  int   (*mkdir)(ufs_errno_t* errno, void* mntdata, const char* dirname);
  int   (*rmdir)(ufs_errno_t* errno, void* mntdata, const char* dirname);
  void* (*opendir)(ufs_errno_t* errno, void* mntdata, const char* dirname);
  int   (*closedir)(ufs_errno_t* errno, void* mntdata, void* dirptr);
  int   (*readdir)(ufs_errno_t* errno, void* mntdata, void* dirptr, char* d_name, int maxlen);
  int   (*unlink)(ufs_errno_t* errno, void* mntdata, const char* pathname);
  int   (*rename)(ufs_errno_t* errno, void* mntdata, const char* oldname, const char* newname);
  int   (*ftruncate)(ufs_errno_t* errno, void* mntdata, void* hdl, ufs_off_t newsize);
  int   (*stat)(ufs_errno_t* errno, void* mntdata, const char* pathname, ufs_stat_t *stat);
  int   (*statvfs)(ufs_errno_t* errno, void* mntdata, const char* pathname, ufs_statvfs_t *stat);
  int   (*chmod)(ufs_errno_t* errno, void* mntdata, const char* pathname, ufs_mode_t mode);
  int   (*utime)(ufs_errno_t* errno, void* mntdata, const char* pathname, ufs_stat_t *times);
  void*       (*open)(ufs_errno_t* errno, void* mntdata, const char* pathname, int flags, ...);
  int         (*close)(ufs_errno_t* errno, void* mntdata, void* hdl);
  ufs_ssize_t (*read)(ufs_errno_t* errno, void* mntdata, void* hdl, void* buf, ufs_size_t count);
  ufs_ssize_t (*write)(ufs_errno_t* errno, void* mntdata, void* hdl, const void* buf, ufs_size_t count);
  ufs_off_t   (*lseek)(ufs_errno_t* errno, void* mntdata, void* hdl, ufs_off_t offset, ufs_seek_t whence);
  int   (*mount)(ufs_errno_t* errno, void* mntdata, const char* name);
  int   (*umount)(ufs_errno_t* errno, void* mntdata, const char* name, int force);
  unsigned strip_rootpath:1;
} ufs_backend_driver_t;
```

UFS

int ufs_mount(const char* name, const void* data,
    const ufs_backend_driver_t* driver);

The specific file system driver bind ufs with the function.

For example: Blunk File System

Blunk File System

```
static const ufs_backend_driver_t _backend = {
  .mkdir          = _mkdir,
  .rmdir          = _rmdir,
  .opendir        = _opendir,
  .closedir       = _closedir,
  .readdir        = _readdir,
  .unlink         = _unlink,
  .rename         = _rename,
  .ftruncate      = _ftruncate,
  .stat           = _stat,
  .statvfs        = _statvfs,
  .open           = _open,
  .close          = _close,
  .read           = _read,
  .write          = _write,
  .lseek          = _lseek,
  .chmod          = _chmod,
  .utime          = _utime,
  .strip_rootpath = 0,
};


int blunk_ufs_mount(const char* name)
{
  return ufs_mount(name, (void*)0, &_backend);

}
```

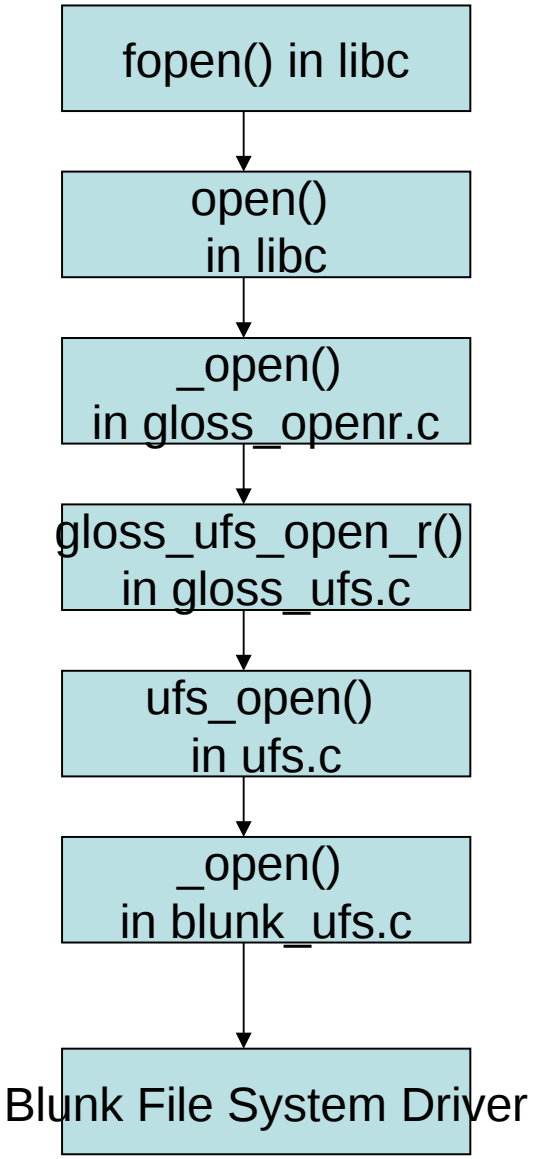
Blunk File System

Blunk file system provide file io functions as follow:

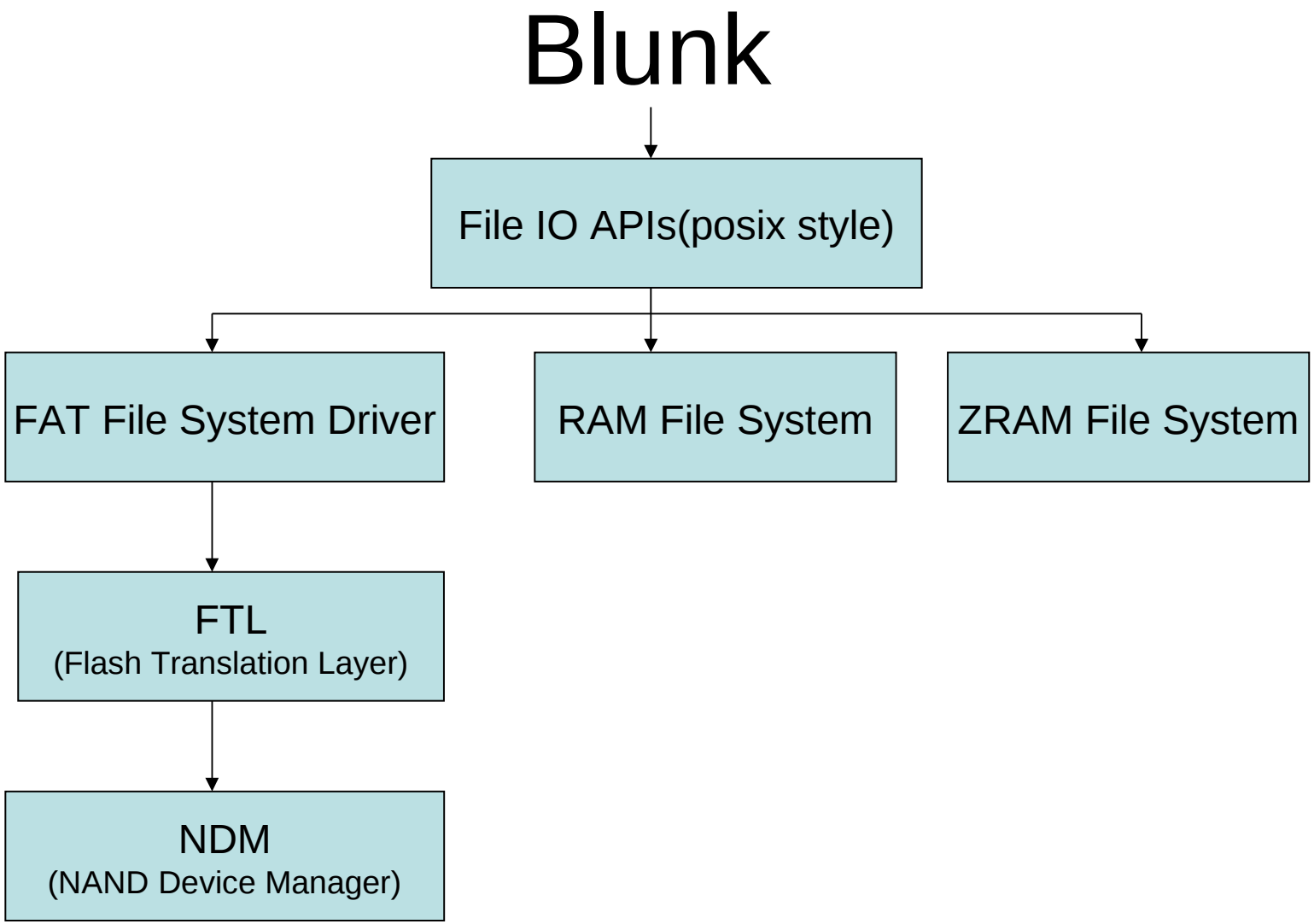
1. system call --- open/read/write/close
2. FILE* --- fopen/fread/fwrite/fclose

These APIs are similar to ones in libc, but you should not invoke them directly.

File IO Call-down Chain



Blunk File System


Blunk
File IO APIs(posix style)
FAT File System Driver
RAM File System
ZRAM File System
FTL
(Flash Translation Layer)
NDM
(NAND Device Manager)

adhesive tape

nand_intf_api.c

(third_party/filesystem/src/blunk/blunk_20112/drivers/marvell/nand_intf_api.c)

marvell_chg_dfc.c

(third_party/filesystem/src/blunk/blunk_20112/drivers/marvell/marvell_chg_dfc.c)

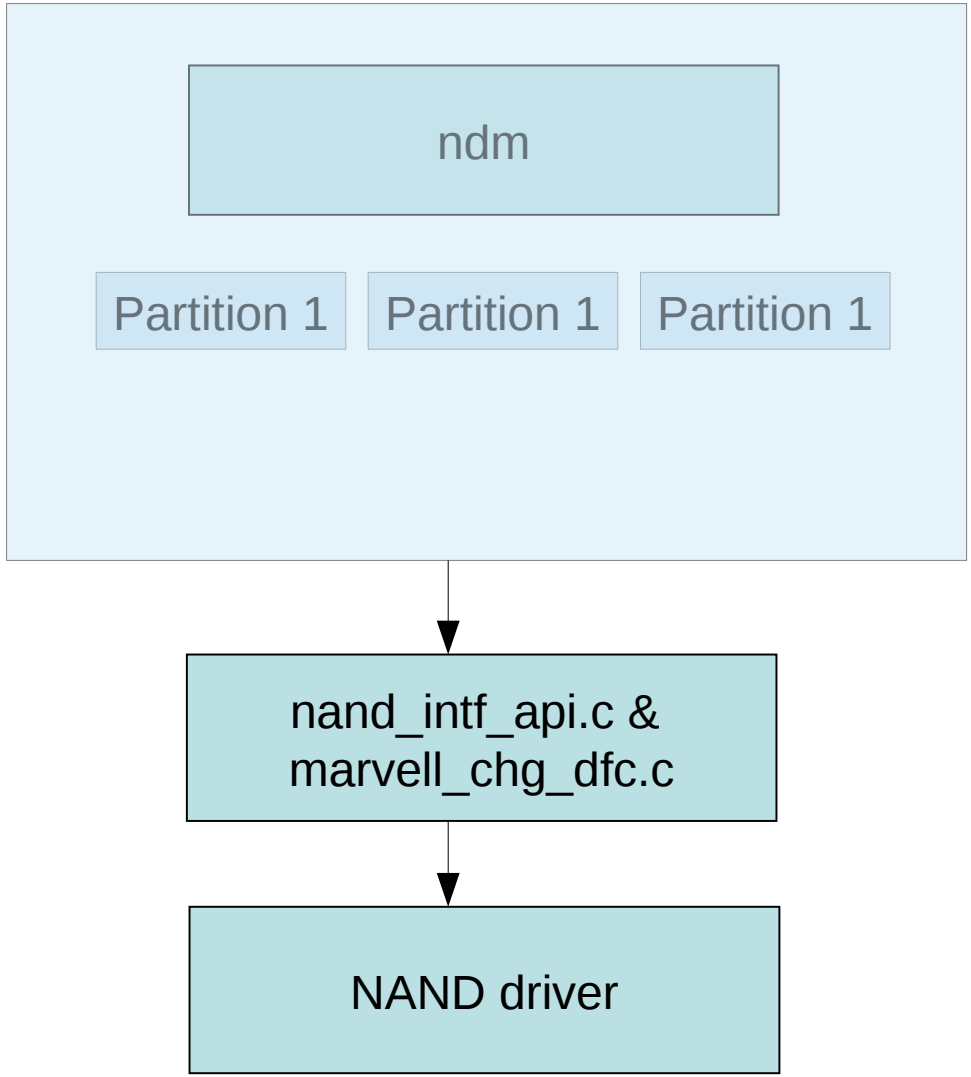
The interfaces in the 2 files is the adhesive tape between NDM of Blunk and NAND driver.

Interfaces

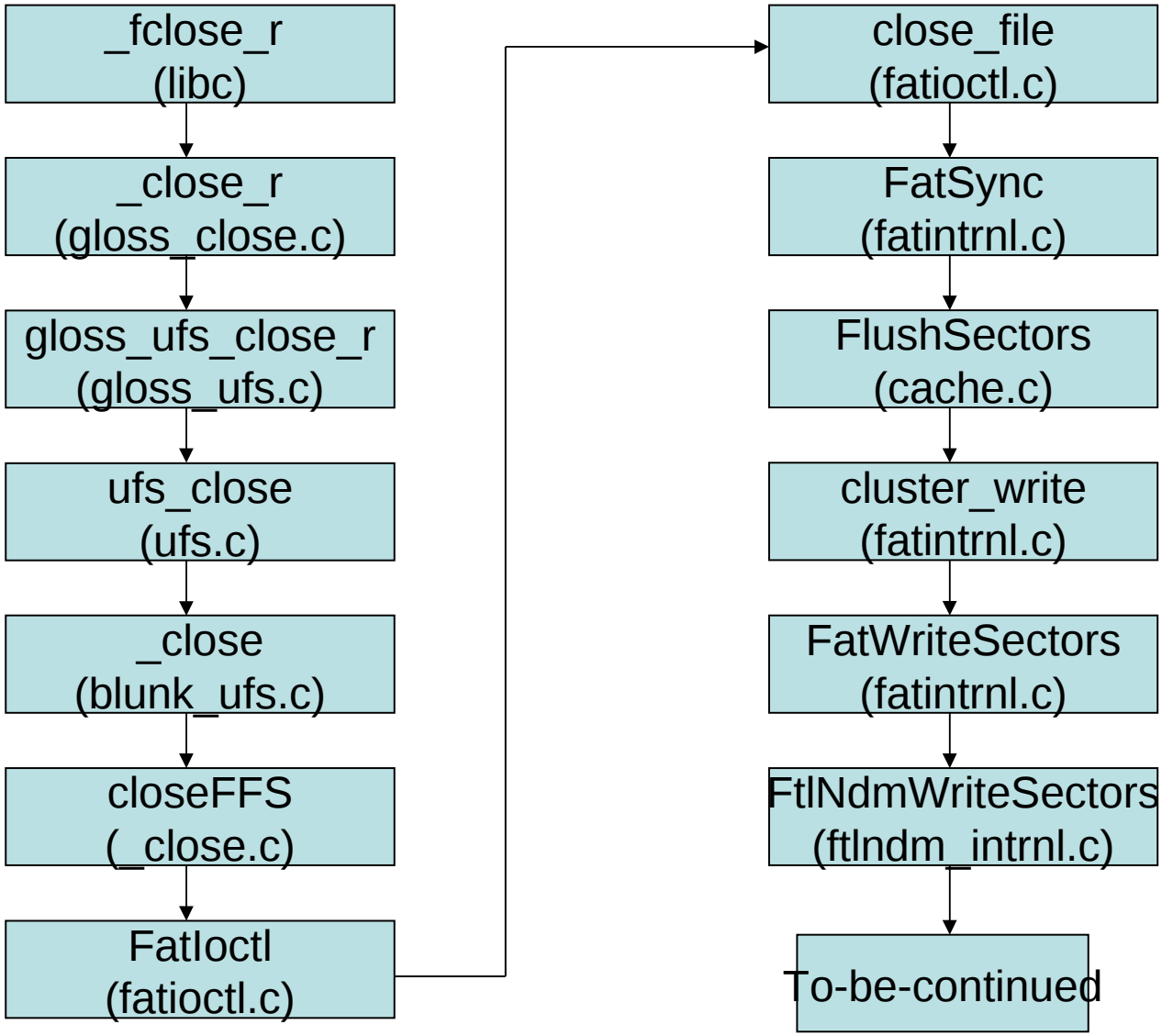
```
typedef struct
{
  ui32 num_blocks;      /* total number of blocks on device */
  ui32 max_bad_blocks;  /* maximum number of blocks that can go bad */
  ui32 block_size;      /* block size in bytes */
  ui32 plane_blocks;    /* number of blocks in a plane */
  ui32 page_size;       /* page data area in bytes */
  ui32 eb_size;         /* page spare area in bytes */
  ui32 flags;           /* option flags */
  ui32 type;            /* type of device */
  void *dev;            /* optional value set by driver */


  /*
  ** Driver Functions
  */
  int  (*write_data_and_spare)(ui32 pn, const ui8 *data, ui8 *spare,
                               int action, void *dev);
  int  (*read_decode_data)(ui32 pn, ui8 *data, ui8 *spare, void *dev);
  int  (*read_pages)(ui32 pn, ui32 count, ui8 *data, ui8 *spare,
                     void *dev);
  int  (*mark_page_dirty)(ui32 pn, ui8 *spare, void *dev);
  int  (*transfer_page)(ui32 old_pn, ui32 new_pn, ui8 *data,
                        ui8 *old_spare, ui8 *new_spare, int encode_spare,
                        void *dev);
  ui32 (*pair_offset)(ui32 page_offset, void *dev);
  int  (*read_decode_spare)(ui32 pn, ui8 *spare, void *dev);
  int  (*read_spare)(ui32 pn, ui8 *spare, void *dev);
  int  (*data_and_spare_erased)(ui32 pn, ui8 *data, ui8 *spare,
                                void *dev);
  int  (*data_and_spare_check)(ui32 pn, ui8 *data, ui8 *spare,
                               int *status, void *dev);
  int  (*erase_block)(ui32 pn, void *dev);
  int  (*is_block_bad)(ui32 pn, void *dev);
#if FS_DVR_TEST
  int  (*chip_show)(void);
  int  (*rd_raw_page)(ui32 p, void *buf, void *vol);
#endif
} NDMDrvr;
```

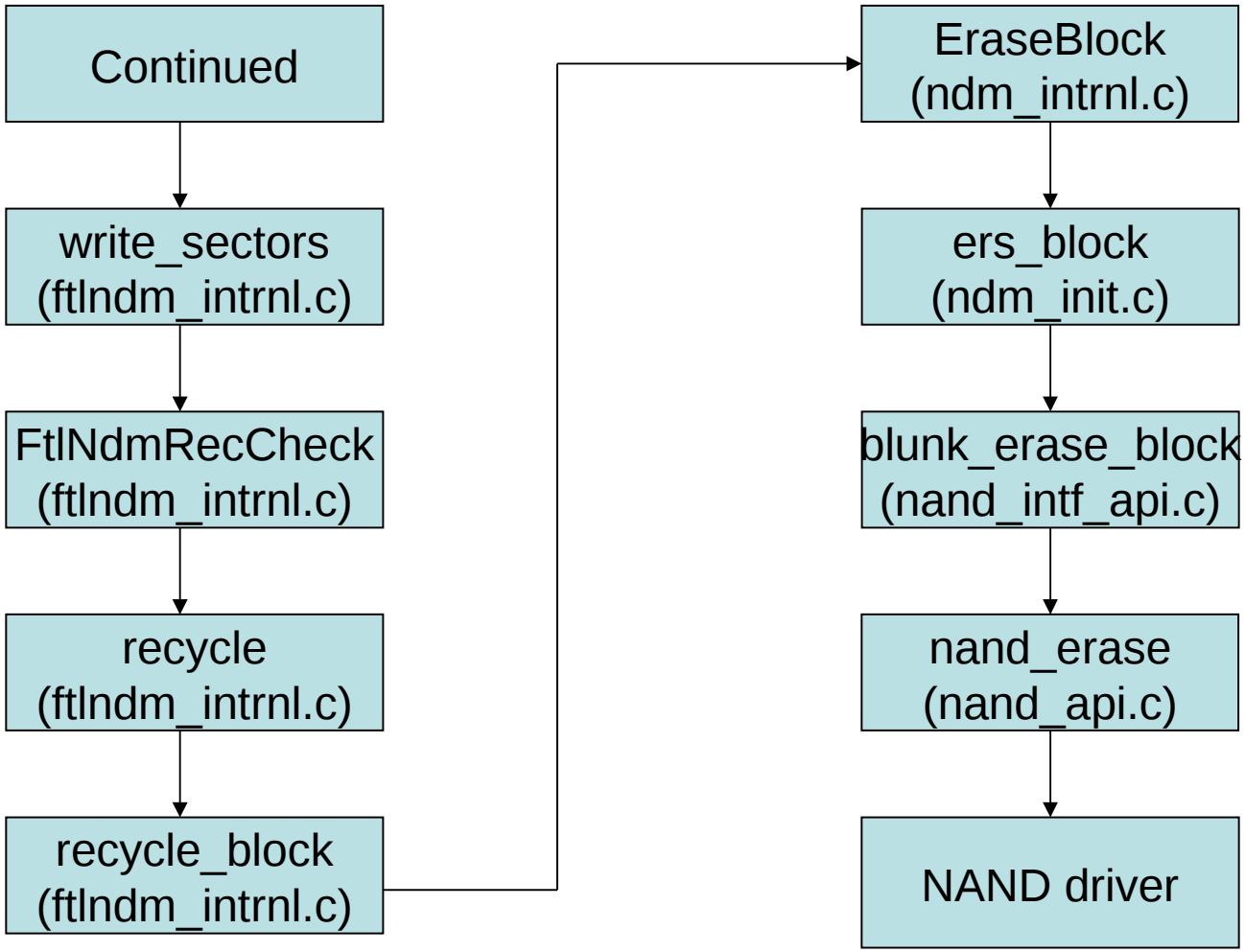
nand_intf_api.c



Whole File IO call down chain



Whole File IO call down chain



Directory

CMD==> ls

/Region1

/Kinoma

/RFA

/data

/tmp

NAND Storage

```
static file_system_partition_list_t partition_list[] =
{
#ifdef HAVE_NAND
   {
      .blocks  = HAVE_EI_PARTITION_BLOCKS,
      .type    = FAT_VOL,
      .name    ="Region1",
      .product_name = EI_PRODUCT_NAME,
      .cdfs   = true,
   },
   {
      .blocks  = HAVE_KINOMA_PARTITION_BLOCKS,
      .type    = FAT_VOL,
      .name    ="Kinoma",
      .product_name = KINOMA_PRODUCT_NAME,
      .cdfs   = false,
   },
   {
      .blocks  = HAVE_FAX_PARTITION_BLOCKS,
      .type    = FAT_VOL,
      .name    ="RFA",
      .product_name = FAX_PRODUCT_NAME,
      .cdfs   = false,
   },
#endif
};
```

/data

/data is implemented by common/datafs module.

It is a pseudo file system like /proc file system in Linux.

It is constructed by the _node_t structures that make into a tree.

You could not write anything to the directory.

But if you want to create a new file system, you could use it as reference.

/data

For example:

/readme.txt

/intpage/mono_config.jpdl

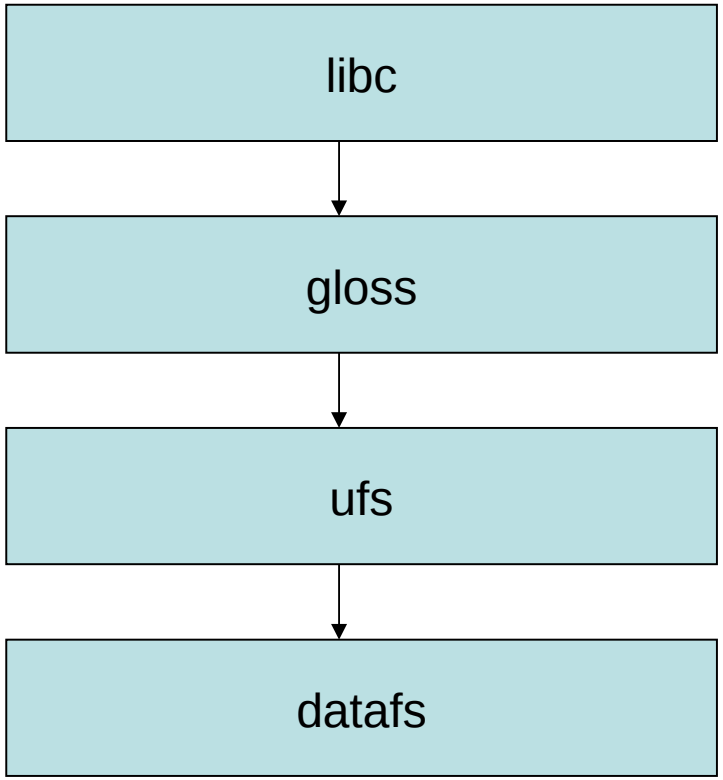
//intpage/jpdl.js

//intpage/mono_config_example.jpdl

/font/default.ttf

You could add file to the directory by datafs_add_file().

/data

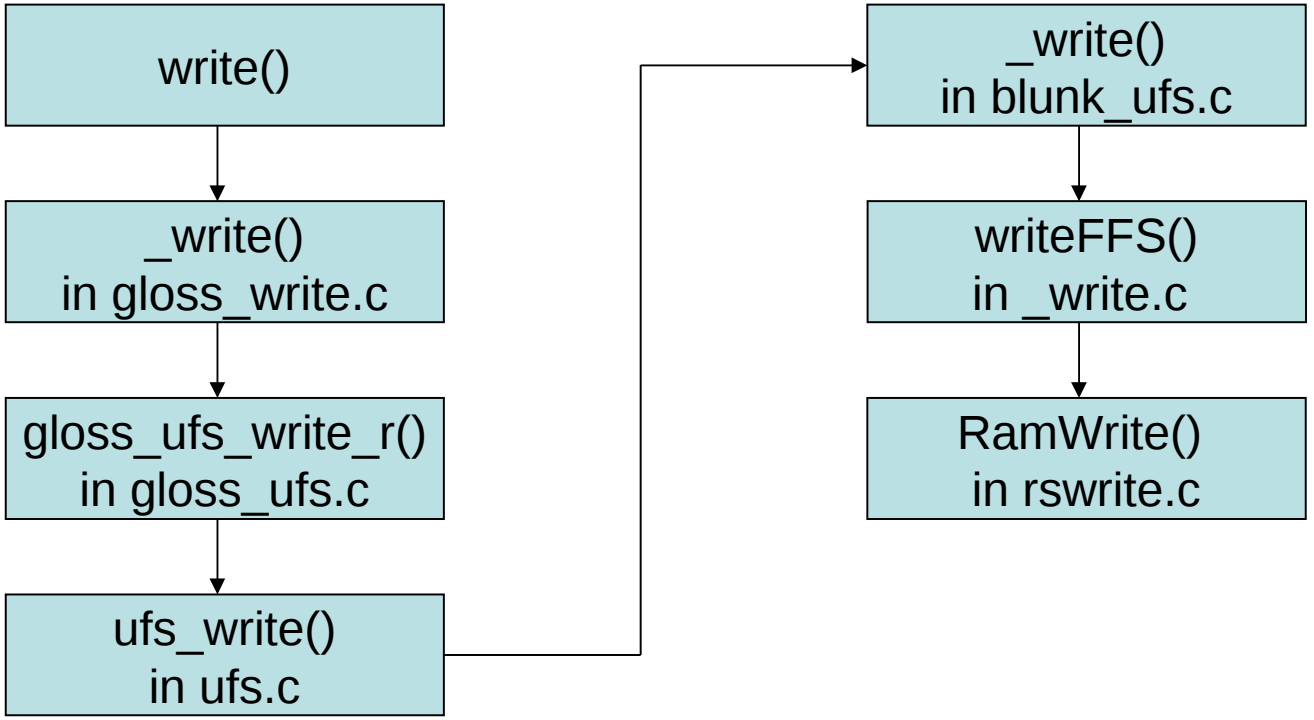


/tmp

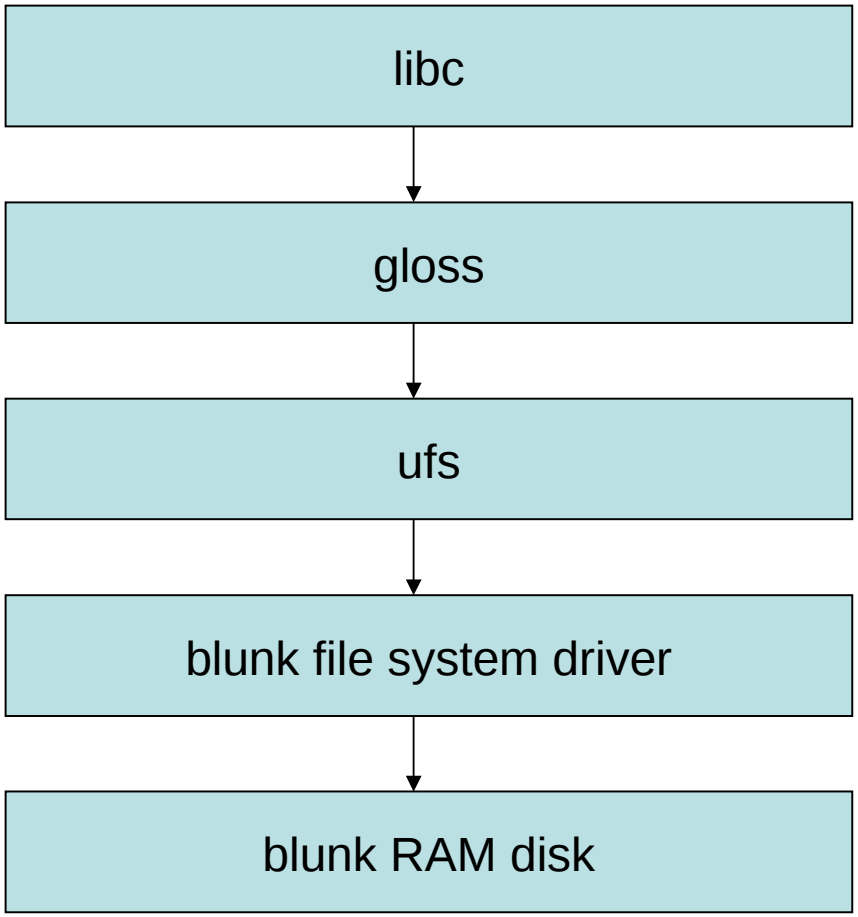
RAM disk

The storage for the directory is from RAM

For example



/tmp

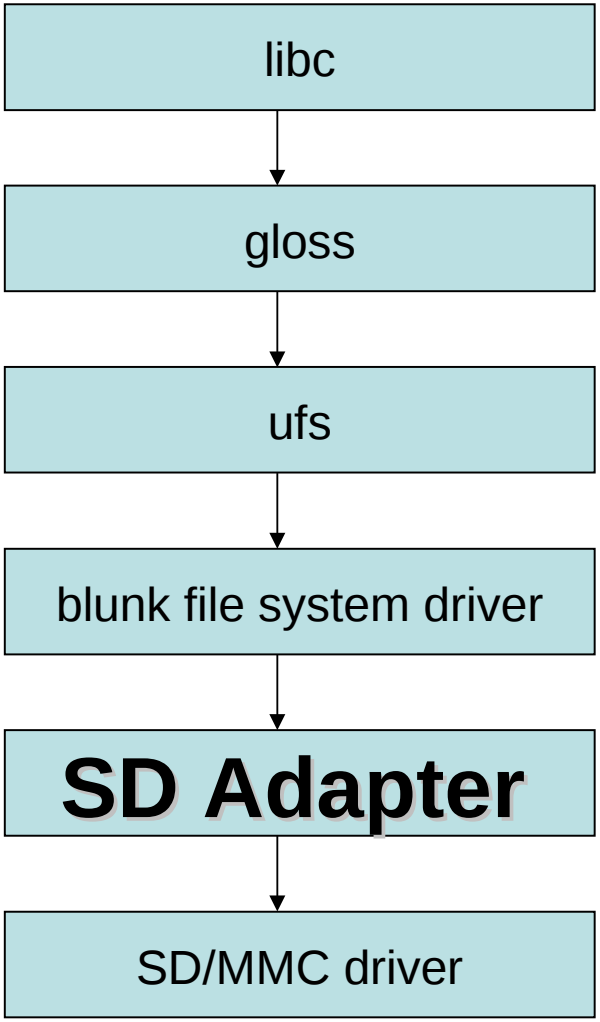


How to support new file system?

Aha! It’s easy.

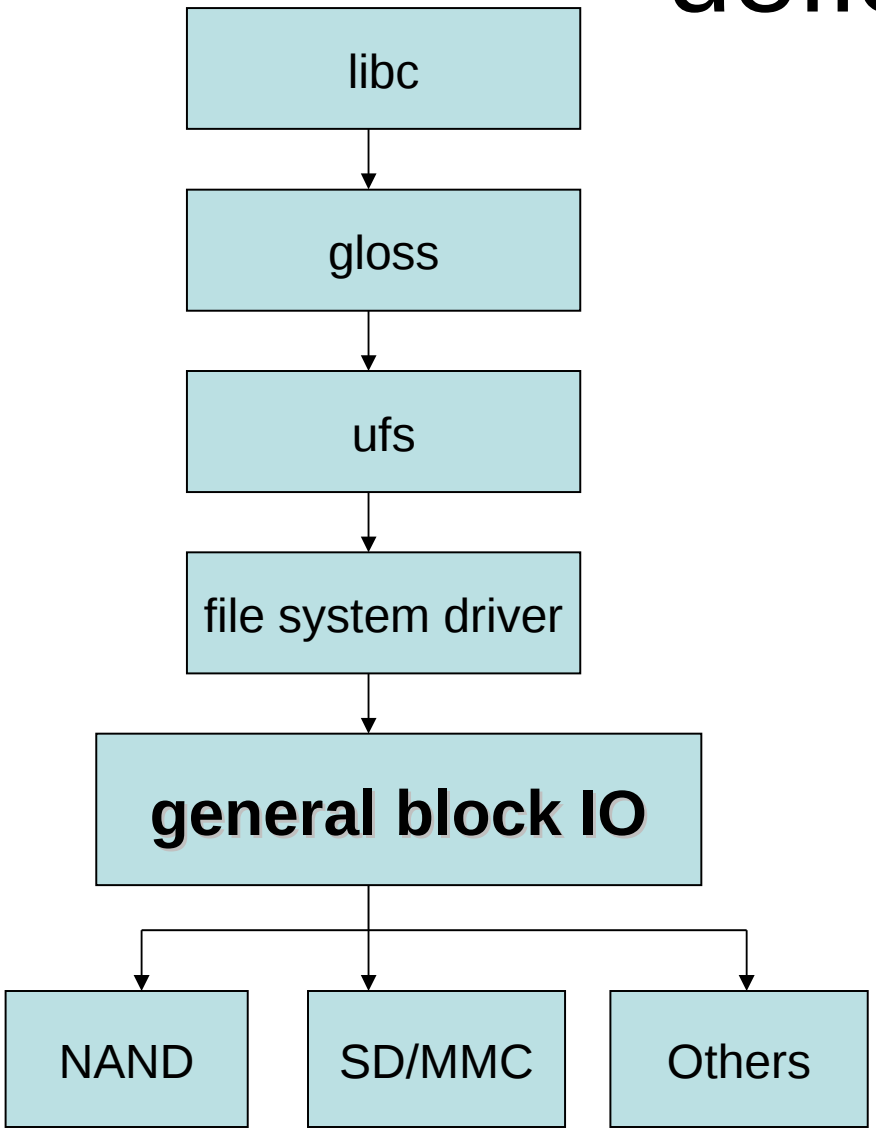
You only need implement all interfaces in ufs_backend_driver_t _backend structure according to the specific file system specification.

How to make SD/MMC support FAT?



We need blunk file system driver, but need not FTL(Flash Translation Layer) and NDM(NAND Device Manager). blunk file system driver implements the FAT file system logic according to FAT specification. When it want to read from or write to the physical storage, we should make it route to SD/MMC driver --- it’s the responsibility of SD Adapter.

deficiency



I think maybe we should create another abstract layer between file system driver and the real storage --- I name it as “general block IO”. All file system drivers should interact with the general block IO layer, and the all real storage driver only care about the interfaces in “general block IO”.

About media_manager

common/media_manager is for “general block io” ?

Hmm…

I think it could not bear the responsibility currently. It’s too simple and only notify the following events:

“media inserted”

“media removed”

“media mounted”

“media mount fail”

But I think media_manager could be a good base for “general block io”.

Debug tips

1. ufs only supports the subset of normal file io, not complete. For example, sync().
2. When the file io function fail, you could not trace it, because there is no libc source files in Marvell SDK. But you could still trace it by setting breakpoint in gloss or ufs module. For example, you want to trace write(), you could set breakpoint on _write() in gloss_write.c or ufs_write() in ufs.c.
3. According to the file IO architecture that Marvell Shanghai have mastered, we could identify which module produce the issue when File IO fail.

FAQ